John von Neumann Institute for Computing

The Mitosis Speculative Multithreaded Architecture

C. Madriles, C. García Quiñones, J. Sánchez,
P. Marcuello, A. González

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 27-38, 2006.

© 2006 by John von Neumann Institute for Computing
Permission to make digital or hard copies of portions of this work for personal or classroom use is granted provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise requires prior specific permission by the publisher mentioned above.

http://www.fz-juelich.de/nic-series/volume33

John von Neumann Institute for Computing

The Mitosis Speculative Multithreaded Architecture

C. Madriles, C. García Quiñones, J. Sánchez,
P. Marcuello, A. González

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 15-26, 2006.

© 2006 by John von Neumann Institute for Computing
Permission to make digital or hard copies of portions of this work for personal or classroom use is granted provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise requires prior specific permission by the publisher mentioned above.

http://www.fz-juelich.de/nic-series/volume33

The Mitosis Speculative Multithreaded Architecture

Carlos Madriles[a], Carlos García Quiñones[a], Jesús Sánchez[a], Pedro Marcuello[a], Antonio González[a]

[a]Intel Barcelona Research Center, Intel Labs - UPC, Barcelona

Abstract

Speculative multithreading (SpMT) increases the performance by means of exploiting speculative thread-level parallelism. In this paper we describe the Mitosis framework, which is a combined hardware-software approach to finding and exploiting speculative thread-level parallelism, even in the presence of frequent dependences between threads. The approach is based on predicting/computing thread input values via software, through a piece of code that is added at the beginning of each thread, the pre-computation slice (p-slice). A p-slice is expected to compute the correct thread input values most of the time, but not necessarily always. Because of that, aggressive optimization techniques can be applied to the slice to make it very short. In this paper, we also describe the microarchitecture that supports this execution model. The main novelty of the microarchitecture is the organization of the register file and the cache memory in order to support multiple versions of each variable and allow for roll-back in case of misspeculation. We also describe a novel compiler approach to identify points where speculative threads are most effectively spawned, and generate the corresponding p-slices for each. We show that the Mitosis microarchitecture-compiler achieves very important speedups for applications that the compiler cannot parallelize by conventional non-speculative approaches, such as the Olden benchmarks.

1. Introduction

Most high-performance processor vendors have now introduced designs that feature processors that can execute multiple threads simultaneously on the same core, either through multithreading [5][15], multiprocessing [20][22] or a combination of the two. The way thread-level parallelism (TLP) is currently being exploited in these processors is through non-speculative threading – all created threads are committed. Basically, there are two main sources of non-speculative TLP: 1) execute different applications in parallel, and 2) execute parallel threads from a single application generated through compiler/programmer support. In the former case, running different independent applications in the same processor provides an increase in throughput (number of jobs finished per time unit) and a reduction in the average response time of jobs over a single-threaded processor. In the latter case, programs are partitioned into smaller threads that are executed in parallel. This partitioning process may significantly reduce the execution time of the parallelized application in comparison with single-threaded execution.

Executing different applications in parallel provides no gain for a single application. On the other hand, partitioning applications into parallel threads may be a straightforward task in regular applications, but becomes much harder for irregular programs, where compilers usually fail to discover sufficient thread-level parallelism, tipically because of the necessarily conservative approach taken by the compiler. This normally results in the compiler including many unnecessary inter-thread communication/synchronization operations, or more likely, concluding that insufficient parallelism exists. Recent studies have proposed the use speculative thread-level parallelism (SpMT). This technique reduces the execution time of applications by executing several speculative threads in parallel.

These threads are speculative in the sense that they may be data and control dependent on previous threads and their correct execution and commitment is not guaranteed. Additional hardware/software is required to validate these threads and eventually commit them.

There are two main strategies for speculative TLP: 1) the use of helper threads to reduce the execution time of high-latency instructions by means of side effects [2][4][10][17][24], and 2) relaxing the parallelization constraints and parallelizing the applications into speculative threads ([1][3][11][18][19][21] among others).

The Helper Thread paradigm is based on the observation that the cost of some instructions severely impact performance, such as loads that miss in cache or branches that are incorrectly predicted. This paradigm attempts to reduce the execution time of the application by using speculative threads to reduce the cost of these high-latency operations.

In speculative parallelization, each of the speculative threads executes a different portion of the program. This partitioning process is based on relaxing the parallelization constraints and allowing the spawning of speculative threads even where the compiler cannot guarantee correct execution. Once the thread finishes, the speculative decisions are verified – if they were correct, then the application has been accelerated. If a misspeculation (control or data) has occurred, then the work done by the speculative thread is discarded and the processor continues with the correct threads.

The Mitosis processor is based on the speculative parallelization approach. The primary distinction from previous works stems from how inter-thread dependences are handled. It is possible to partition a program into enough parallel threads such that there are few or no dependences between them [19]. However, for most programs it is necessary to create threads where there are control/data dependences across these partitions to fully exploit the available parallelism. The manner in which such dependences are managed critically affects the performance of the SpMT processor [11]. Previous approaches have used hardware communication of produced register values [9], explicit synchronization [6][21], and value prediction [12] to manage these dependences.

In contrast, in the Mitosis framework we propose a new, software approach to manage both data and control dependences among threads. We find that this both produces values earlier than the hardware-assisted value-passing approaches, and more accurately than hardware value prediction approaches (due to the use of the actual application code). Each speculative thread is prepended with a speculative pre-computation slice (p-slice) that precomputes live-ins, while executing in a fraction of the time of the actual code that produces those live-ins.

In this paper, we present the Mitosis processor, a hardware/software approach to effectively exploit speculative TLP. The Mitosis framework is composed of a compiler that partitions the applications into speculative threads and a speculative multithreaded processor that is able to manage multiple speculative threads. It also provides novel mechanisms to track and manage the different versions of the memory and the register values for the speculative threads. The overall Mitosis processor framework is presented, including the compiler architecture, the hardware architecture, and several novel aspects of each. Some very promising early performance results are presented as well. This study focuses on applications that sophisticated parallelizing compilers cannot parallelize. Performance results reported by Mitosis processors show an average speed-up of about 2.2x for a subset of the Olden benchmark suite and 1.75x over a configuration that models perfect L1 level caches.

The rest of the paper is organized as follows: Section 2 presents the execution model. The different components of the Mitosis processors are described: the compiler in Section 3 and the microarchitecture in Section 4. Section 5 presents the performance evaluation of this architecture. Finally, Section 6 summarizes the main conclusions of the work.

2. Execution Model of the Mitosis Processor

Mitosis is a framework that combines hardware and software techniques to exploit speculative TLP. The Mitosis compiler is responsible for partitioning the application into speculative threads. It also plays an important role in dealing with data dependences since it generates the code to compute the live-in values of the speculative threads. The Mitosis processor architecture provides the hardware support for executing speculative threads and detecting any possible misspeculation.

Figure 1 shows the execution model of the Mitosis processor. Programs are partitioned into speculative threads statically with the Mitosis compiler. The partitioning process done by the compiler is described in Section 3. It basically explores the application to insert spawning pairs, that is, pairs of instructions made up of the point where the speculative thread will be spawned (the spawning point, or SP for short) and the point where the speculative thread will start its execution (the control quasi-independent point, or CQIP for short) [13]. The Mitosis compiler also computes the p-slice of each spawning pair that predicts the input values of the speculative thread.

This new binary obtained with the Mitosis compiler is executed on the Mitosis processor. Applications run on a Mitosis processor in the same way as on a conventional superscalar processor. However, when a spawn instruction is found, the processor looks for the availability of a free context (or thread unit). On finding a free one, the p-slice of the corresponding speculative thread is assigned to be executed at that thread unit. The p-slice ends with an unconditional jump to the CQIP, thereby starting the execution of the speculative thread. If no free thread unit is available when the spawn instruction is executed, the system looks for a speculative thread that is more speculative (further in sequential time) than the new thread we want to spawn. If any is found, the most speculative one is cancelled and its thread unit is assigned to the new thread.

Threads in Mitosis processors are committed in program order. The thread executing the oldest instructions in program order is non-speculative, whereas the rest are speculative. When any running thread reaches the CQIP of any other active thread, it stops fetching instructions until it becomes the non-speculative thread. Then, a verification process checks that the next speculative thread has been executed with the correct input values. If the speculation has been correct, the non-speculative thread is committed and its thread unit is freed. Moreover, the next speculative thread becomes the non-speculative. If there is a misspeculation, the next speculative thread and all its successors are squashed, and the non-speculative thread continues executing the instructions beyond the CQIP.

In Mitosis processors, the spawning process is highly general. Any speculative or non-speculative thread can spawn a new thread upon reaching an SP. Moreover, speculative threads can be spawned out of program order, that is, threads can be created in a different order than they will be committed.

3. Mitosis Compiler

The Mitosis compiler has been developed on top of the Open Research Compiler (ORC)[8]. The ORC infrastructure is a research IPF compiler that produces code with a quality similar to that of a production compiler. The different Mitosis modifications have been implemented in the back-end of the compiler after all the optimizations and before bundle formation. These modifications include the following highly coupled steps: 1) identification of spawning pairs, and 2) generation and optimization of the p-slice. We will describe these two steps below.

3.1. Spawning Pair Identification

The partitioning of applications into speculative threads is performed by means of the identification of spawning pairs. A spawning pair represents two instructions: the spawning point (SP) and

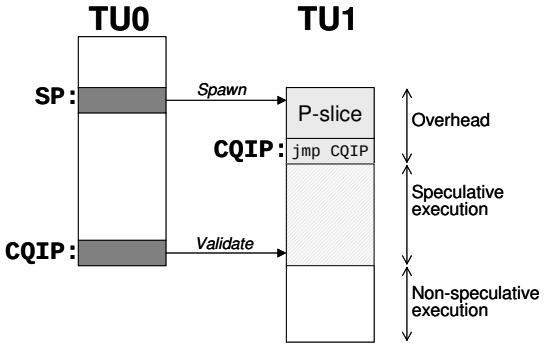

Figure 1. Mitosis execution model

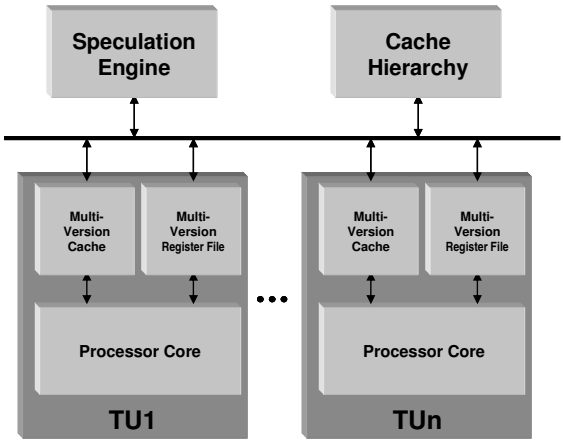

Figure 2. Mitosis processor μarchitecture

the control quasi-independent point (CQIP). This pair of instructions could be formed by any pair of instructions in the program, but there are some requirements they should meet in order to provide some benefit. Examples of these requirements are control and data independence, load balance issues, thread length, etc.

In this work, we propose a static mechanism that selects the spawning pairs by means of a cost model that estimates the expected benefit of any potential spawning pair. This model also considers the probability/cost of misspeculations and the interaction with other spawning pairs selected.

The Mitosis compiler uses edge profiling information, as extracted by many conventional compilers, to initially obtain a list of possible spawning pair candidates. This initial set of candidates is obtained pruning the whole set by a certain control independence and a minimum distance between the SP and the CQIP. These candidates have the SP and the CQIP in the same routine.

Once this step has been performed, the p-slice for each of these potential spawning pairs is calculated. The way slices are generated is described in the next subsection. Candidates are pruned again depending on the size of the p-slice. Large p-slices imply fewer overlapped instructions. Thus, those spawning pairs whose p-slice size, relative to the expected size of the body of the thread, is higher than a certain threshold are discarded.

Next, a synthetic trace of the program is generated from the edge profile information and a greedy algorithm is applied to get the best set of spawning pairs. In this algorithm, the pair that performs the best individually based on the cost model is selected among all the candidate spawning pairs . Taking into account this inserted thread, this process is repeated until the increasing benefit between two consecutive iterations is lower than a certain threshold.

3.2. Slice Generation

Given a spawning pair, a p-slice is the subset of the instructions executed between the spawning point and the control quasi-independent point needed for computing values used beyond the control quasi-independent point. Therefore, the first step to get the p-slice is to determine the thread live-in values. The compiler examines the code following the CQIP, but only for as many instructions as the average distance between the spawning point and the control quasi-independent point (because once the spawning thread reaches the CQIP and verifies the spawned thread, all values are available non-speculatively). Then, for these thread input values, it is determined which of them are produced

between the spawning pair and the corresponding data and control dependence graph for only those values is built. Those instructions that are in the sub-graphs of the thread live-ins are selected to be inserted in the slice. Finally, the slice ends with an unconditional branch to the CQIP.

The identification of the thread input values is relatively straightforward for register values, but harder for memory values. To detect thread input memory values, the Mitosis compiler uses an improved version of the memory dependence profiling.

Subroutine calls within the spawning pair that belong to the sub-graph of the thread input values are also included in the p-slice. However, subroutines that perform system calls are not included and then spawning pairs that require a system call in the p-slice are not considered for selection.

A key observation for generating p-slices is that they do not need to be correct. This is because the hardware, as described in the next section, will validate them and squash the thread when they are incorrect. Therefore, the compiler includes aggressive, sometimes unsafe, optimizations such as thread and slice branch pruning and memory dependence speculation. Complete information regarding the compiler support and the applied p-slice optimizations can be found at [14].

4. Mitosis Processors

The Mitosis processor has a multi-core design similar to an on-chip multiprocessor (CMP), as shown in Figure 2. Each thread unit executes a single thread at a time, and it is similar to a superscalar core. Each speculative thread may have a different version for each logical register and memory location. The different versions are supported by means of a local register file and a local memory per thread unit. In this section, the most relevant components of the microarchitecture are described, including the Multi-Version Register File and the Multi-Version Memory.

4.1. Spawning process

A speculative thread starts when any active thread fetches a spawn instruction in the code. The spawn instruction triggers the allocation of a thread unit, the initialization of some registers, and the ordering of the new thread with respect to the other active threads. These tasks are handled by the Speculation Engine.

To initialize the register state, the spawn instruction includes a mask that encodes which registers are p-slice live-ins. Those registers included in the mask are copied from the parent thread to the spawned one. On average, we have observed that just 6 registers need to be initialized for our benchmarks.

Any active thread is allowed to spawn a new thread when it executes a spawn instruction. Additionally, speculative threads can be spawned out of program order, that is, a speculative thread that would be executed later than another thread if executed sequentially can be spawned and executed in reverse order in a Mitosis processor. Some previous studies have shown that out-of-order spawning schemes have a much higher performance potential than in-order approaches [1] [11].

It is also necessary to properly order the spawned thread with respect to the rest of the active speculative threads. The order among threads will determine where a thread is going to look for values not produced by itself. Akkary and Driscoll [1] propose a simple mechanism in which the new spawned thread is always assumed to be the most speculative. On the other hand, Marcuello and González [11] propose an order predictor based on the previous executions of the speculative threads. In this work, this latter scheme is used since it provides better hit ratio (98% with the configuration described in Section 5).

4.2. Multi-Version Register File

To achieve correct execution and high performance, the architecture must simultaneously support the following, seemingly conflicting, goals: a unified view of all committed register state, the co-existence of multiple versions of each register, register dependences that cross thread units, and a latency similar to a single local register file.

This support is provided by the Mitosis multi-version register file, shown in Figure 3. As can be observed, the register file has a hierarchical organization. Each thread unit has its own Local Register File (LRF) and there is a Global Register File (GRF) for all the thread units. There is also a table, the Register Versioning Table (RVT) that has as many rows as logical registers and as many columns as thread units, that tracks which thread units have a copy of that logical register.

When a speculative thread is spawned in a thread unit, some register values are copied from the parent thread. These registers are encoded in the spawn instruction, as described above. Slices have the characteristic that they are not more nor less speculative than the parent thread; in fact, they execute a subset of instructions of the parent thread so the speculation degree is the same. Therefore, it needs to access the same register versions as the parent thread would see at the spawning point (while executing in a different thread unit).

When a thread requires a register value, the LRF is checked first. If the value is not present, then the RVT is accessed to determine which the closest predecessor thread that has a copy of the value. If there are no predecessors that have the requested register, then the value is obtained from the GRF.

There is an additional structure used for validation purposes: the Register Validation Store (RVS). When a speculative thread reads a register for the first time and this value has not been produced by the thread itself the value is copied into this structure. Additionally, those register values generated by the p-slice that have been used by the speculative thread are also inserted in the RVS. When this thread is validated, the values in the RVS are compared with the actual values of the corresponding registers in the predecessor thread. Doing so we ensure that values consumed by the speculative thread would have been the same in a sequential execution. Because we explicitly track values consumed, incorrect live-ins produced by the slice that are not consumed do not cause misspeculation.

Finally, when the non-speculative thread commits, all the modified registers in the LRF are copied into the GRF. An evaluation of the performance impact of the Multi-Version Register File design has been done. On average, more than 99% of the register accesses are satisfied from the LRF for the benchmarks evaluated. Thus, the average perceived latency for register access is essentially equal to the latency of the LRF, meeting the goals for our register file hierarchy.

4.3. Multi-Version Memory System

Similar to the register storage, the memory system provides support for multi-versioning, that is, it allows different threads to have different values for the same memory location. As shown in Figure 4, each thread unit has its own L0 and L1 data caches, which are responsible for maintaining the speculative values, since speculative threads are not allowed to modify main memory. Moreover, three additional structures are needed at each thread unit – the Slice Buffer (SB), the Old Buffer (OldB) and the Replication Cache (RC). Finally, there is a global L2 cache shared among all the thread units which can only be updated by the non-speculative thread, and a centralized logic to handle the order list of the different variables, the Version Control Logic (VCL).

The architecture of this memory system is inspired by the Speculative Versioning Cache (SVC) proposed by Gopal et al. [7] with notable extensions to handle p-slices. As a summary, the Mitosis memory subsystem contains the following novel features: support for p-slice execution and the replication cache. This section focuses on these new features.

A load in a p-slice needs to read the value of that memory location at the SP, while a load in the

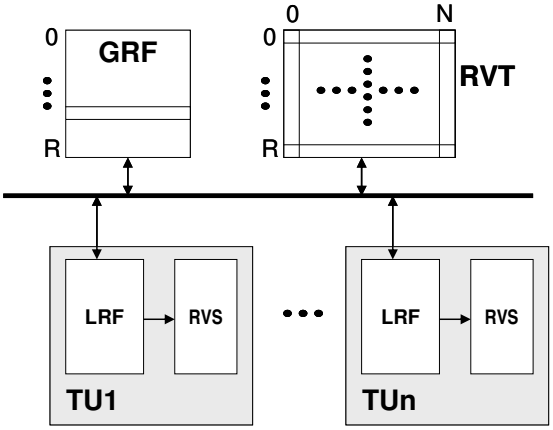

Figure 3. Multi-version register file

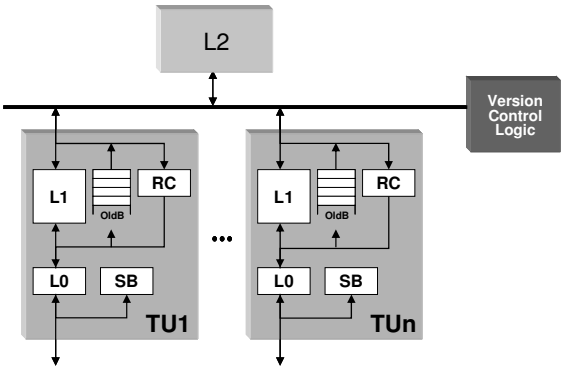

Figure 4. Multi-version memory system

thread needs to see the value at the CQIP, if it has not been produced by itself. For this reason, during the execution of a p-slice the processor needs to have an exact view of the machine state of the parent at the time of the spawn instruction. Therefore, when a thread performs a store and any of its children is still executing its p-slice, the value of that memory location needs to be saved before overwriting it since its children may later require that value. The buffers used for storing the values that are overwritten while a child is executing a p-slice are referred to as Old Buffers. Each thread unit has as many Old Buffers as direct child threads are allowed to be executing a p-slice simultaneously. Thus, when a speculative thread that is executing the p-slice performs a load to a memory location, it first checks for a local version at its local memory. In case of a miss, it checks in its corresponding Old Buffer from the parent thread. If the value is there, then it is forwarded to the speculative thread. Otherwise, it looks for it at any less speculative thread cache. When a speculative thread finishes its slice, it sends a notification to its parent thread to deallocate the corresponding Old Buffer. Finally, it is possible that a thread finishes its execution and some of its children are still executing the p-slice. Then, those Old Buffers cannot be freed until these threads finish their corresponding p-slices.

The values read during the execution of the p-slice have to be marked in some way to avoid being read by any other thread, and mistaken for state expected to be valid across CQIPs. To prevent a more speculative thread from reading an incorrect value, a new bit is added to the SVC protocol that is referred to as Old Bit. When a thread that is executing the slice performs a load from the parent Old Buffer, the value is inserted into the local cache with the Old Bit set. Then, when a more speculative thread requests this value, if it finds the Old Bit set it knows that the value stored in that cache may be potentially old and is ignored. Finally, when the slice finishes, all the lines of the local cache with the Old Bit set are invalidated.

Slice Buffers are used to store the values computed by the live-in p-slice in order to validate if the speculative thread is being executed with the correct input values. When a thread is allocated to a thread unit, the Slice Buffer is reset and all the stores performed during the execution of the slice go directly to Slice Buffer, bypassing the cache. Each entry of the Slice Buffer contains an address, a value, a read bit and a valid bit. Thus, when the slice finishes and the speculative thread starts its execution, every time a value is read from memory, the Slice Buffer is checked first. If the value is there, the read bit is set and the value copied to cache. When the thread becomes the non-speculative one, all the values that have been consumed from the Slice Buffer have to be checked for their

correctness. Thus, those entries that have their read bit set are sent to the previous non-speculative thread to validate their values.

When threads only exploit samethread memory locality, cache miss rates would be extreme. Preliminary experiments showed that the miss ratio of the L1 cache was very high for the speculative threads. This is due to the fact that when a thread commits, all the lines of the local cache set its commit bit to defer the traffic burst to main memory. As a result, every newly spawned thread begins with a completely empty cold cache. The impact of this feature could be reduced by introducing to the SVC protocol the stale bit as was proposed elsewhere [7]. However, this memory model has the problem of very poor locality exploitation. If all the active threads consume the same memory line, values have to travel from one thread unit to another every time. Mitosis solves that with the Replication Cache (RC). This small cache works as follows: when a thread performs a store, it has to send a bus request to know if any more speculative thread has performed a load on that address. We use this request to send the value and store it in the Replication Cache of all the threads that are more speculative as well as in the free thread units. Thus, when a thread performs a load, L1 and the Replication Cache are checked simultaneously. If the value requested is not in L1 but it is in the Replication Cache, the value is moved to L1 and supplied to the thread unit. This simple mechanism prevents the thread units from starting with cold caches as well as taking advantage of locality.

4.4. Thread Validation

A thread finishes its execution when it reaches the starting point of any other active thread, that is, the CQIP. At this point, if the thread is nonspeculative, it validates the next thread. Otherwise, it waits until it becomes the non-speculative one. The first thing to verify is the order. The CQIP found by the terminating thread is compared with the control quasi-independent point of the following thread in the thread order (as maintained by the order predictor). If they are not the same, then an order misspeculation has occurred and the following thread and all its successors are squashed.

If the order is correct, then the thread input values used by the speculative thread are verified. These comparisons may take some time, depending on the number of values to validate. We have observed that on average, for our workloads, a thread validation requires to check less than 1 memory and about 5 register values for this validation. Note that only memory values produced by the slice and then consumed by the thread (values read from the slice buffer) need to be validated when the previous thread finishes. Other memory values consumed by the thread are dynamically validated as soon as they are produced through the versioning protocol described above. If no misspeculations are detected, the non-speculative thread is committed and the next thread becomes the nonspeculative one. The thread unit assigned to the finished thread is freed, except when there is a child thread that is still executing the p-slice since it may require values available in the Old Buffer.

5. Experimental Framework

The performance of the Mitosis processors was evaluated through a detailed execution-driven simulation. The Mitosis compiler has been implemented on top of the ORC compiler to generate IPF code. The Mitosis processor simulator models a research Itanium CMP processor with 4 hardware contexts based upon SMTSIM [23]. The main parameters considered are shown in Table 1. The numbers in the table are per thread unit.

To evaluate the potential performance of the Mitosis architecture, a subset of the Olden suite [16] has been used. The benchmarks used are the bh, em3d, health, mst and perimeter, with an input set that on average executes around 300M instructions. Statistics in the next section correspond to the whole execution of the programs. Different input data sets have been used for profiling and

Mitosis processor configuration

Fetch, in-order issue and commit bandwidth	2 bundles (6 instructions)	**Order Predictor size**	16K-entry
Reorder buffer	512 instructions	**Branch Predictor**	Local Gshare
I-Cache	64KB	**Local Register File**	1 cycle
L0-Cache	4-way 16 KB – hit: 1 cycle	**Global Register File**	256-entry / 6 cycles
L1-Cache	4-way 1 MB – hit: 3 cycles	**Spawn overhead**	5 cycles
Crossfeed latency	3 cycles	**Validation overhead**	15 cycles
Replication cache	4-way 16 KB	**Slice buffer**	1K-entry – hit: 1 cycle
L2-Cache (shared)	4-way 8 MB – hit: 6 cycles; miss: 250 cycles	**Old buffer**	3 – 128-entry each

simulation. The rest of the suite has not been considered due to the recursive nature of the programs. Up to this point, the Mitosis compiler is not able to extract speculative TLP in recursive routines. This feature will be targeted in future work.

Olden benchmarks have been chosen since they are pointer intensive programs and automatic parallel compilers are unable to extract TLP. To corroborate this, we have compiled the Olden suite with the Intel C++ production compiler which produces parallel code. Almost none of the code was parallelized by this compiler.

5.1. Performance Figures

Table 2
Characterization of the Olden benchmarks

Benchmarks	#Spawned threads	Thread size	Slice size	%Slice / thread	Thread live-ins	%Squashes
bh	422	15543.0	196.5	1.3	4.4	0.7
em3d	396638	422.1	9.0	2.1	1.0	0.3
health	198497	1112.7	41.6	3.7	2.7	26.9
mst	1367114	271.4	5.8	2.1	2.3	0.8
perimeter	493725	576.8	24.0	4.2	3.6	1.0
AMEAN	**491279.2**	**3585.2**	**55.4**	**2.7**	**2.8**	**6.0**

Statistics corresponding to the characterization of the speculative threads are shown in Table 2. The last row shows the arithmetic mean for the evaluated benchmarks. The first column shows the number of spawned threads by benchmark and the second column the average number of speculative instructions executed by the speculative threads. It can be observed that bh spawns the fewest number of threads but their average size is about 30 time larger than for the rest of benchmarks. On the other hand, mst spawns the most but their average size is the lowest. The third column shows the average dynamic size of the slices and the fourth column the relationship between the sizes of the speculative threads and their corresponding slice. This percentage is consistently quite low for all the studied benchmarks and on average represents less than 3%. The fifth column shows the average number of thread input values that are computed by the slice, that is, on average it is only necessary to compute 3 values to execute the speculative threads. Finally, the right-most column represents the average number of squashed threads. For all the benchmarks, this percentage is rather low except for health where almost 1 of every 4 threads is squashed. We have observed that for this particular benchmark,

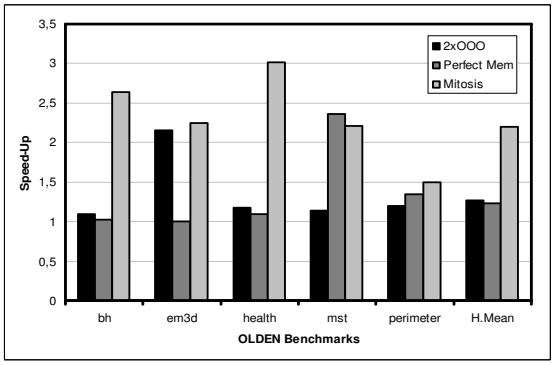

Figure 5. Speed-up over single thread

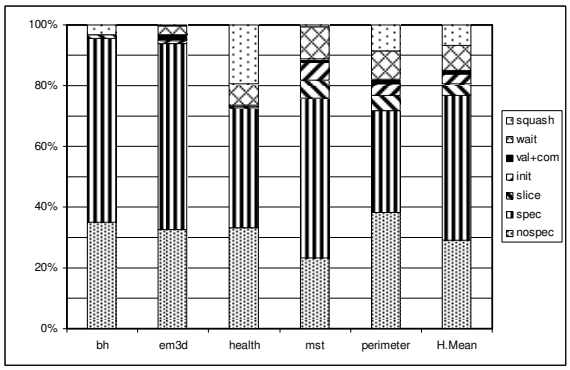

Figure 6. Time breakdown for Mitosis

memory dependences for the profiling and simulated inputs are significantly different, which result in many memory dependence misspeculations.

Figure 5 shows the speedups of the Mitosis processor over a superscalar in-order processor with about the same resources as one Mitosis thread unit with no speculative threading. For comparison, we also show the speedup of a more aggressive processor, with twice the amount of resources (functional units), twice the superscalar width and out-of-order issue (with no speculative threading), and a processor with perfect first-level cache (an aggressive upper limit to the performance of helper threads that target cache misses).

It can be observed that the Mitosis processor achieves an average speed-up close to 2.2x over single-threaded execution, whereas the rest of the configurations provide much lower performance. Perfect memory achieves a speed-up of just 1.23x and the more aggressive out-of-order processor only provides a 1.26x speed-up. Some results worth further discussion are the similar performance achieved by the Mitosis processor compared with the outof- order double-wide core in em3d and compared with the perfect memory model in mst. In the former case, ILP is quite abundant in this benchmark, which could be additionally exploited with more complex Mitosis configurations (with out-of-order thread units). In mst, the performance of the memory system is quite low (for a single-threaded execution, the miss ratio in L1 cache is higher than 70% and close to 50% in L0). Even so, Mitosis is surprisingly competitive with the perfect memory configuration (within 8%). In summary, we find Mitosis mirrors an aggressive superscalar when ILP is high, an unattainable memory subsystem when memory parallelism is high, and outperforms both when neither ILP nor memory parallelism is high.

Figure 6 shows the time breakdown for the execution of the different benchmarks in the Mitosis processors. As expected, most of the time the thread units are executing useful work (the sum of the non-speculative and the speculative execution). On average, this percentage is almost the 80% of the time the thread units are working and higher than 90% for bh and em3d. The overhead added by this execution model represents less than 20% for these benchmarks. The most significant part of this overhead comes from the wait time. This time stands for the time a thread unit has finished the execution of a speculative thread but it has to wait until becoming non-speculative to commit. The other components of the overhead are the slice execution, the initialization overhead, and the validation and commit overhead. It is worth noting that the execution of the slices only corresponds to 4%. Finally, the top of the bars shows the average time thread units are executing incorrect work, that is, threads that are squashed. This percentage is only 8% overall, mostly due to health, where

the overhead is almost 20%. In this case, most of the squashes are due to memory violations and the cascading effect of the squashing mechanism. Recall, however, that health still maintains a 3x speedup despite these squashes.

These results strongly validate the effectiveness of the execution model introduced in this paper (precomputation slice based speculative multithreading) in meeting the Mitosis design goals: high performance, resulting from high parallelism (low wait time), high spawn accuracy (very low squash rates), and low spawn and prediction overhead (very low slice overheads).

6. Conclusions

In this work, we have presented and evaluated the Mitosis framework, which exploits speculative TLP through a novel scheme to deal with interthread data dependences. This model is based on predicting via software the thread live-ins. It does so by means of inserting a piece of code in the binary that speculatively computes the values at the starting point of the speculative thread. This code, referred to as a p-slice, is built from a subset of the code after the spawn instruction and before the beginning of the speculative thread. A key feature of Mitosis processors is that p-slices do not need to be correct, which allows the compiler to use aggressive optimizations when generating them.

An efficient mechanism to partition the code into speculative threads is presented. This mechanism looks into the code to detect which parts of the program will provide the highest benefit, taking into account possible misspeculations, overheads, and load balancing. Moreover, some compiler optimization techniques have been presented in order to reduce the weight of the slices in the speculative threads.

The key microarchitecture components of Mitosis processors have been presented: (1) hardware support for the spawning, execution, and validation of p-slices allows the compiler to create slices with minimal overhead, (2) a novel multi-version register file organization supports a unified global register view, multiple versions of register values, transparent communication of register dependences across processor cores, all with no significant latency increase over traditional register files, and (3) a memory system that support multiple versions of memory values, and introduces a novel architecture that pushes values towards threads that are executing future code, to effectively mimic the temporal locality available to a single-threaded processor with a single cache.

Finally, the results obtained by the Mitosis processor with 4 thread units for a subset of the Olden benchmarks are impressive. It outperforms the single-threaded execution by 2.5x, and more than 1.75x compared with a double-size out-of-order processor, and over a perfect memory model. These results confirm that there are large amounts of TLP on codes that can be extracted by speculative techniques such as those of Mitosis on codes that cannot be parallelized by conventional approaches.

Acknowlegdments

We would like to thank Peter Rundberg (currently at Gridcore, Sweden), Hong Wang and John Shen (at Intel Labs, Santa Clara) and Dean Tullsen (at UC San Diego) for his valuable collaboration in this work. Also, we would like to thank the ORC team for their support in the compiler implementation. This work has been partially supported by the Spanish Ministry of Education and Science under contract TIN2004-03072 and Feder funds.

References

[1] H. Akkary and M.A. Driscoll: A Dynamic Multithreading Processor. Procs. of the 31st Int. Symp. on Microarchitecture. 1998

[2] R.S. Chappel et al.: Simultaneous Subordinate Microthreading (SSMT). Procs. of the 27th Int. Symp. on Computer Architecture. 2000

[3] L. Codrescu and D. Wills: On Dynamic Speculative Thread Partitioning and the MEM-Slicing Algorithm. Procs. of the Int. Conf. on Parallel Architectures and Compilation Techniques. 1999

[4] J.D. Collins et al: Speculative Precomputation: Long Range Prefetching of Delinquent Loads. Procs. of the 28th Int. Symp. on Computer Architecture. 2001

[5] K. Diekendorff: Compaq Chooses SMT for Alpha. Microprocessor Report(December). 1999

[6] M. Franklin and G.S. Sohi: The Expandable Split Window Paradigm for Exploiting Fine Grain Parallelism. Procs. of the 19th Int. Symp. on Computer Architecture, pp. 58-67, 1992.

[7] S. Gopal, T.N. Vijaykumar, J.E. Smith and G.S. Sohi: Speculative Versioning Cache. Procs. of the 4th Int. Symp. on High Performance Computer Architecture. 1998

[8] http://ipf-orc/sourceforge.net

[9] V. Krishnan and J. Torrellas: Hardware and Software Support for Speculative Execution of Sequential binaries on a Chip-Multiprocessor. Int. Conf. on Supercomputing. 1998

[10] C. Luk: Toleraing Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. Procs. of the 28th Int. Symp. on Computer Architecture. 2001

[11] P. Marcuello: Speculative Multithreaded Processors. Ph.D. Thesis, Universitat Politecnica de Catalunya. 2003

[12] P. Marcuello, J. Tubella and A. González: Value Prediction for Speculative Multithreaded Architectures. Procs. of the 32nd. Int. Conf,. on Microarchitecture. 1999

[13] P. Marcuello and A. González: Thread-Spawning Schemes for Speculative Multithreaded Architectures. Procs. of the 8th Int. Symp, on High Performance Computer Architectures. 2002

[14] C. García et. al. : Mitosis Compiler: an Infrastructure for Speculative Threading Based on Pre-Computation Slices. Proc. of the Int. Symp. on Programming Language Design and Implementation. 2005

[15] T. Marr et al.: Hyper-threading Technology Architecture and Microarchitecture. Intel technology Journal, 6(1). 2002

[16] A. Rogers, M. Carliste, J. Reppy and L. Hendren: Supporting Dynamic Data Structures on Distributed Memory Machines. ACM Trans on Programming Languages and System (March). 1995

[17] A. Roth and G.S. Sohi: Speculative Data-Driven Multithreading. Procs. of the 7th. Int. Symp. on High Performance Computer Architecture. 2001

[18] G.S. Sohi, S.E. Breach and T.N. Vijaykumar: Multiscalar Processors. Procs. of the 22nd Int. Symp. on Computer Architecture. 1995

[19] J. Steffan and T. Mowry: The Potential of Using Thread-level Data Speculation to Facilitate Automatic Parallelization. Procs. of the 4th Int. Symp. on High Performance Computer Architecture. 1998

[20] S. Storino an dJ. Borkenhagen: A Multithreaded 64-bit PowerPC Commercial RISC Processor Design. Procs. Of the 11th Int. Conf. on High Performance Chips. 1999

[21] J.Y. Tsai and P-C. Yew: The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation. Procs. of the Int. Conf. on Parallel Architectures and Compilation Techniques. 1995

[22] M. Tremblay et al.: The MAJC Architecture, a synthesis of Parallelism and Scalability. IEEE Micro, 20(6). 2000

[23] D. M. Tullsen, S.J. Eggers and H.M. Levy: Simultaneous Multithreading: Maximizing On-Chip Parallelism. Procs. of the 22nd Int. Symp. on Computer Architecture. 1995

[24] C.B. Zilles and G.S. Sohi: Execution-Based Prediction Using Speculative Slices. Procs. of the 28th Int. Symp. on Computer Architecture. 2001